Aix*Marseille université
Initiative d'excellence
Année 2018-19

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Semaine 2 : TP 7

Diviser pour régner

1 Rotation d'une image d'un quart de tour en espace constant

Une tâche relativement classique à effectuer sur des images numériques consiste à effectuer une rotation d'un quart de tour, par exemple dans le sens horaire. Considérons ici des images bitmap, c'est-à-dire des tableaux bi-dimensionnels de *pixels*, chacun de ces pixels étant un tableau tri-dimensionnel pour représenter le niveau de rouge, vert et bleu, chacun entre 0 et 255. En Python, on peut utiliser la bibliothèque `imageio` pour charger le contenu d'une image dans un tableau `numpy`. Par exemple, pour charger l'image `fibre-optique.png` donnée sur Ametice, on peut utiliser le code

```
import imageio
import numpy
picture = imageio.imread("fibre-optique.png")
```

En utilisant les possibilités de la bibliothèque `numpy`, on peut facilement effectuer une symétrie par rapport à l'axe vertical d'une image :

```
def horizontal_flip(image):
    """renvoie l'image obtenue par symétrie selon l'axe vertical"""
    # création d'une image vide qui sera utilisée comme résultat de la fonction
    image_sym = numpy.empty_like(image)
    height, width = image.shape[0], image.shape[1]
    # on calcule l'image cible en fonction de l'image source
    for j in range(width):
        numpy.copyto(image_sym[:, j, :], image[:, (width - 1) - j, :])
    return image_sym
```

Noter l'utilisation de la fonction `copyto` de la bibliothèque `numpy` qui permet de copier dans le tableau en premier argument le contenu du tableau en second argument : cela a donc a priori le même comportement que l'affectation `image_sym[:, j, :] = image[:, (width - 1) - j, :]`. En fait, pas tout à fait puisque la fonction `copyto` (comme la plupart des fonctions arithmétiques de la bibliothèque `numpy`) permet le *broadcast*, ce qui veut dire que `numpy` s'adapte si le tableau à copier est en fait une valeur unique, auquel cas il la copie dans toutes les cases du tableau donné en premier argument : `numpy.copyto(u, 2)` permet de placer un 2 dans toutes les cases du tableau `u`, contrairement à `u = 2` qui remplace simplement le contenu de la variable `u` par 2.

On peut utiliser la fonction `horizontal_flip` puis sauver le contenu de la nouvelle image dans un autre fichier qu'on peut ensuite lire avec un programme externe :

```
picture_flipped = horizontal_flip(picture)
imageio.imsave("fibre-optique-horizontal-flip.png", picture_flipped)
```

Question 1. Modifier l'algorithme afin d'obtenir une fonction renvoyant la rotation d'un quart de tour dans le sens horaire d'une image **carrée** (de sorte que la hauteur et la largeur de l'image à renvoyer sont les mêmes que celles de l'image originelle).

Question 2. En supposant que la fonction `copyto` ait une complexité de l'ordre du nombre d'éléments à copier, quelle est la complexité de cette fonction ?

Question 3. Ces fonctions ne sont pas *en place*, mais créent une nouvelle image, sans modifier l'image originelle. Comment modifier le code de la fonction de rotation pour qu'elle soit *en place*, sans changer l'ordre

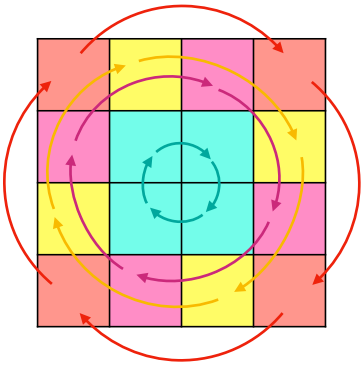

FIGURE 1 – Rotation en place

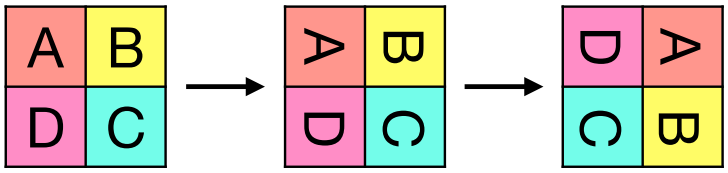

FIGURE 2 – Rotation par un algorithme diviser pour régner

de grandeur de complexité ? On pourra remarquer qu'une telle rotation peut s'obtenir en composant des cycles de longueur 4, comme représenté dans la Figure 1.

En pratique, cette solution n'est en fait souvent pas très efficace dès lors que les images sont un peu grandes, c'est-à-dire dès lors que le processeur doit utiliser des méthodes de *cache* pour récupérer et modifier des zones de l'image. L'inefficacité vient du fait que cette technique nécessite de modifier des valeurs *éloignées* forçant le processeur à recalculer le même cache de nombreuses fois. Une autre méthode est possible, en utilisant une opération peu coûteuse proposée par les cartes graphiques. Elle est illustrée dans la Figure 2.

On remarque qu'appliquer une rotation d'un quart de tour dans le sens horaire d'une image carrée de largeur 2^k (pour k entier) peut se faire en :

- **divisant** l'image en 4 quadrants, résultant en quatre images carrées de largeur 2^{k-1}
- sur lesquels on **règne** en appliquant récursivement la rotation (sauf si chaque cadran est en fait un unique pixel)
- puis à laquelle on applique un cycle de permutation des quadrants dans le sens horaire pour **combiner** les résultats.

On suppose que la carte graphique dispose d'une fonction `blit` permettant de copier une image carrée de largeur n dans une autre image carrée de largeur n, avec une complexité $f(n)$. Le cycle de permutation des quadrants peut alors être réalisé à l'aide de 5 appels à la fonction `blit`, en utilisant une zone mémoire tampon (l'algorithme ne sera donc pas en place).

Question 4. Il est aisé de simuler en Python une telle fonction `blit` avec la fonction `copyto` de la bibliothèque `numpy`. Quelle est alors la complexité de l'algorithme de rotation dans ce cas, sur une image carrée de largeur 2^k ?

Question 5. Supposons désormais que la carte graphique sait implémenter la fonction `blit` avec une complexité $f(n) = \mathcal{O}(n)$. Que devient la complexité dans ce cas-là ?

Question 6. Implémenter l'algorithme diviser pour régner en Python de manière récursive, en utilisant `copyto` pour implémenter la fonction `blit`.

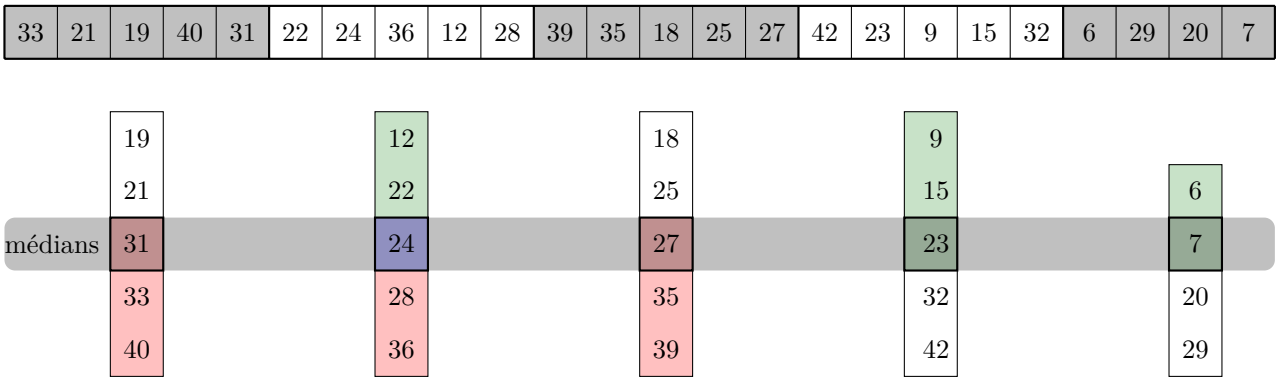

FIGURE 3 – Recherche de l'élément médian

2 Recherche de l'élément médian d'un tableau

Il est souvent très utile de calculer l'élément médian d'une collection de données, pour évaluer par exemple le salaire médian d'une population ou la luminosité médiane des pixels d'une image. Supposons donc donné une liste d'éléments $[e_0, e_1, \ldots, e_{n-1}]$ distincts d'un ensemble ordonnée (des entiers ou des chaînes de caractères, par exemple). L'élément médian est l'élément e_i tel que $|\{e_j \mid e_j \leq e_i\}| = \lceil n/2 \rceil$, c'est-à-dire l'élément qui a autant d'éléments plus grands que de plus petits (à un près). On appellera *rang* d'un élément e la quantité $|\{e_j \mid e_j \leq e\}|$. On peut généraliser le problème pour rechercher l'élément de rang k pour $1 \leq k \leq n$.

Question 7. Proposer une solution simple pour recherche l'élément de rang k de complexité $O(n \log n)$, pour une valeur de k fixée.

On cherche désormais à faire mieux, c'est-à-dire à obtenir une solution en temps linéaire $O(n)$. Suivons pour cela la même idée que celle du tri rapide :

— on choisit un pivot dans la liste,

— puis on partitionne la liste en deux listes, celle des éléments inférieurs ou égaux au pivot notée ℓ_{inf} et celle des éléments strictement supérieurs notée ℓ_{sup}.

Question 8. En supposant qu'on sache calculer l'élément de rang k' (pour toute valeur de k' possible) des deux sous-listes, en déduire comment combiner les résultats pour trouver l'élément de rang k de la liste initiale.

Il reste à expliciter comment on choisit le pivot. Il est possible de montrer qu'un choix randomisé permet d'obtenir un algorithme dont la complexité moyenne est en $\mathcal{O}(n)$. En pratique, c'est ce choix randomisé qu'on privilégie. Cependant, il est intéressant de savoir si l'on peut obtenir cette même complexité avec un algorithme *déterministe* (c'est-à-dire qui ne s'autorise pas de tirages aléatoires). C'est possible avec la méthode suivante :

— on commence par grouper les éléments de la liste par paquets de 5 (un des paquets peut contenir moins de 5 éléments, si n n'est pas multiple de 5)

— on établit ensuite la liste $\ell_{medians}$ des médians de chaque paquets, par exemple en triant chaque paquet

— puis on recherche récursivement le médian de $\ell_{medians}$ qu'on choisit comme pivot. ‘

On illustre cette méthode en Figure 3.

Question 9. Montrer que le rang de l'élément pivot est compris entre $3n/10 - 2$ et $7n/10 + 2$.

Question 10. Quelle est la complexité de la recherche du pivot, si on ne compte pas la complexité de l'appel récursif à l'algorithme de calcul du médian ?

Question 11. Notons $T(n)$ la complexité dans le pire des cas pour la recherche du k-ième élément dans une liste de taille n, avec $1 \leq k \leq n$ quelconque. Montrer que pour n suffisamment grand

$$T(n) \leq T(n/5) + T(7n/10 + 2) + \mathcal{O}(n)$$

Question 12. En déduire la complexité de cet algorithme en conjecturant le résultat grâce à une méthode de substitution, puis en le vérifiant par récurrence.